EVALUATION OF HARDWARE-BASED DATA FLOW INTEGRITY

A Thesis

by

ABHIJITH REDDY RACHALA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Shaoming Huang
Committee Member,	Narasimha Annapareddy
Head of Department,	Miroslav M. Begovic

August 2019

Major Subject: Computer Engineering

ABSTRACT

Computer security is a very critical problem these days, as it has widespread consequences in case of a failure of computer systems security, like desktop machines, mobile phones, tablets and Internet of Things (IoT) devices. Usually, attackers try to find vulnerabilities in the target systems and by exploiting these vulnerabilities, they launch an attack, thereby achieving their malicious goal.Software data attacks modify the intended control/data flow in a program that is unprotected. Control data attacks are executed by exploiting buffer overflows or string vulnerabilities to overwrite a return address, a function pointer or some other information about control data. Non-control data attacks exploit similar vulnerabilities to overwrite security critical data without changing the intended control-flow in the program. Data flow integrity ensures that the flow of data in a program at runtime is permitted by the data flow graph.

The main objective of the thesis is to implement a hardware-based data flow integrity technique and check for vulnerabilities on a target application. This implementation is achieved by referencing a data flow graph against which the runtime data flow of a program is checked. DFI checking is integrated into existing processor with most changes in hardware going to the load/store unit and the arithmetic unit. In gem5, this is realised by modifying source code of the simulator at instruction level to monitor each load/store instruction on the target application and check if there are any data flow violations and check the overhead caused by the modification of gem5 source code to integrate DFI checking with existing CPU models on gem5. From experiments results, we measured the performance overhead to be up to 14.5%. We also roughly estimate the extra hardware required for this implementation on real hardware.

DEDICATION

To my parents

ACKNOWLEDGMENTS

It is my honor and privilege to have pursued my graduate studies at Texas A&M University. I am grateful to many people for their support during this journey. Firstly, I would like to express my sincere gratitude to my advisor, Dr. Jiang Hu for steering my endeavor in academic research through his guidance, patience and understanding. His trust and encouragement helped me to think beyond the normal conventions from time to time and to gave me the freedom to experiment with different ideas, which proved very beneficial in carrying out my research. I would sincerely thank my co-advisor, Dr. Jeff Huang, for his persistent optimism and motivation. I feel fortunate to receive directions from both of my advisors without which this thesis would be impossible. It was truly an honor to have research advisors and mentors like them. I would like to thank Dr. Narasimha Reddy for being a part of my thesis committee and providing continuous constructive feedback on my thesis.

I am very thankful to my colleague in the project, Lang Feng, for sharing his knowledge and ideas for my project. His constant inputs and feedback helped me whenever I was stuck in the project. I am also grateful to Erick Carvajal and Gino Chacon for assisting me when I started off with gem5 and sharing their experiences working with gem5, this helped me grasp the gem5 environment quicker.

I wish to thank the Department of Electronics and Computer Engineering at Texas A&M University for providing the opportunity and resources to fulfill my academic ambition.

I am obliged to all my friends for helping me keep my life in context. I would like to thank my family for their undetering support, encouragement and faith in me.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jiang Hu, and Professor Narasimha Reddy from the Department of Electrical and Computer Engineering (ECE), and Professor Jeff Huang of the Department of Computer Science and Engineering (CSE).

Tools used in the research, namely gem5 and SVF tools are open source tools developed by third parties. Usage of these tools have been duly cited in the thesis.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was partly supported by a scholarship from the ECE department at Texas A&M University.

NOMENCLATURE

DFI	Data Flow Integrity
CFI	Control Flow Integrity
SVF	Static Value Flow
CDI	Core Debug Interface
CPU	Central Processing Unit
ISA	Instruction Set Architecture
CPI	Cycles Per Instruction
DFG	Data Flow Graph
CFG	Control Flow Graph
RDS	Reaching Definition Set
RDT	Reaching Definition Table
TPIU	Trace Port Interface Unit
PC	Program Counter
LUT	Look Up Table
HDFI	Hardware-Assisted Data Flow Isolation
ALM	Adaptive Logic Module

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Security has been extensively researched in the field of general-purpose computers and communication systems, which lead to many advances in cryptographic algorithms and security protocols. Even though such advances in security measures provide a very concrete reason for securing computing systems, recent trends have made it very clear that most attacks target weaknesses in a system's implementation [1]. The 2005 U.S. President's Information Technology Advisory Committee (PITAC) report stated: "Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year" [2].

A U.S. Department of Homeland Security 2006 Draft, "Security in the Software Lifecycle," states the following: *The most critical difference between secure software and insecure software lies in the nature of the processes and practices used to specify, design, and develop the software . . . correcting potential vulnerabilities as early as possible in the software development lifecycle, mainly through the adoption of security-enhanced process and practices, is far more cost-effective than the currently pervasive approach of developing and releasing frequent patches to operational software* [3].

Irrespective of how rigorous the verification of software at development phase is, it is very difficult to detect all errors during development. Therefore, formal and informal checking of specified properties against executing systems or programs is a topic that has been explored from quite some time (prominent examples of this are dynamic typing in software, or fail-safe devices or watchdog timers in hardware). System security can be compromised either through the execution of programs that originate from untrusted or unknown sources, or through the corruption of binaries while they are being downloaded or stored on a system [4]. Attacks on software try to exploit buffer overflows and format string vulnerabilities in order to write data to unintended locations. Techniques like Control Flow Integrity (CFI) and Data Flow Integrity (DFI) are enforced to prevent these attacks. DFI computes a data-flow graph using static analysis, and it instruments the

program to ensure that the flow of data at runtime is allowed by the data-flow graph [5]. In this thesis, a hardware based approach to DFI is explored and implemented on gem5. The rest of the thesis is organized as follows. In this section (Section 1), we further give a brief introduction about runtime verification, the importance of security in systems and the need for techniques like CFI and DFI. In Section 2, some previous work in this domain are introduced. In Section 3, a detailed background is given which is necessary to understand the idea of this thesis better. In Section 4, the main idea of the thesis, that is, the hardware implementation of DFI on gem5 is explained in detail. Section 5 introduces the experimental setup and discusses the experiments run and results obtained. Finally, a summary is provided in Section 6.

1.1 Runtime Verification

Runtime verification is a computing system analysis and execution approach based on extracting information from a running system and using it to detect and possibly respond to observed behaviors which satisfy or violate intended properties [6] . Runtime verification specifications are usually expressed in trace predicate formalisms, such as finite state machines, regular expressions, context-free patterns, linear temporal logics, etc., or extensions of these. This allows for a less ad-hoc approach than normal testing [6]. However, any monitoring mechanism in an executing system is considered runtime verification, including verifying against test oracles and reference implementations.

Runtime verification can be used for a wide variety of applications, like security and safety policy monitoring, verification, debugging, testing, validation, profiling, fault protection, recovery etc. Runtime verification evades the complexity of traditional formal verification techniques, like model checking and theorem proving, by analyzing only one or a few execution traces and by working directly with the actual system. Therefore, it scales up relatively well and gives more confidence in the analysis results but at the expense of less coverage. Moreover, runtime verification can be made an integral part of the target system, monitoring and guiding its execution during deployment [6].

Formal or informal checking of specified properties against executing systems or programs is

a topic that has been explored from quite some time, whose precise roots are hard to identify. The termi runtime verification was formally introduced through the name of a 2001 workshop aimed at addressing problems at the boundary between formal verification and testing. Writing test cases for large code bases is a very tedious and time consuming task. Moreover, some errors may not be detected during development. Early contributions to automate verification have been made at the NASA Ames Research Center by Klaus Havelund and Grigore Rosu to archive high safety standards in spacecrafts, rovers and avionics technology. They proposed a tool to verify specifications in temporal logic and to detect race conditions and deadlocks in Java programs by analyzing single execution paths [6].

The field of runtime verification methods can be broadly classified into:

- The system can be monitored during the execution itself (online) or after the execution, for example in form of log analysis (offline).
- The verifying code is integrated into the system or is provided as an external entity.
- The monitor can report violation or validation of the desired specification.

1.2 Security

As mentioned earlier, recent trends have shown that most attacks target weaknesses in a system's implementation [1]. Therefore, it is now a commonly agreed fact that a secure system implementation is as critical to a system's overall security as the strength of the theoretical security measures employed [1]. As a result, in recent years there has been an increasing awareness that security at various stages of system design process, including system architecture and hardware/software implementation, needs to be considered. Execution of programs/applications originating from unknown or untrusted sources and corruption of binaries when they are being downloaded or stored on a system are the reasons a system's security is compromised. Executing a code that has been obtained from a trusted source also doesn't ensure safe execution. Even a trusted code can be hijacked at run time, therefore the original code may not be malicious by intent but it can be manipulated by attackers and can result in destructive or harmful behavior. There are multiple ways to

execute a security attack. Software security exploits take advantage of weaknesses in code [operating system , middleware, applications] that is already present in the system. Amongst these, buffer overflow attacks, which exploit the lack of bounds checking in C/C++ programs, have emerged as one of the most common forms of security violations [1]. Many embedded systems are mobile devices with small form factors that may be passed around in the hands of adversaries for a period of time is sufficient to launch such attacks. The outcome of such attacks is especially dangerous when they are used to subvert programs that have special privileges, e.g., access to sensitive data or system resources [1]. Even in the embedded system domain, a recent trend has been a sharp increase in embedded software content in order to support increasing end-user functionality and performance requirements [4]. With complexity of software increasing and times-to-market getting reduced, many software bugs and vulnerabilities may go undetected during the design phase. With increased connectivity, embedded systems are now able to automatically download and install software, which exposes these systems to malicious programs and in turn making them easy targets for attackers.

1.3 Monitoring Techniques

In this section, monitoring techniques like CFI and DFI are explained and we give an idea about why such checking is needed. Control-flow integrity is a general term for computer security techniques which prevent a wide variety of malware attacks from redirecting the flow of execution of a program [7]. CFI is used to monitor and control instruction flow transitions like branches and jumps and make sure they adhere to the intended reference design. Through this software execution can be stopped from executing malicious code which could result in corruption of an application or a system. The CFI security policy checks that the control flow of a program must follow a path of defined by the a Control-Flow Graph (CFG), which is determined prior to the program execution (using static analysis).

CFI stops control attacks by guaranteeing that the control flow remains within the control-flow graph intended by the design. Every instruction that is the target of an allowed control-flow transaction is assigned a unique identifier, and to make sure that only valid targets are allowed checks are

inserted before control-flow instructions [8]. Any program typically has two types of control-flow transfers: direct or indirect. Direct transfers have a fixed target and they do not need any enforcement checks. However, indirect transfers, such as branch instructions, function calls and returns, take a dynamic target address as argument. As the target address could be modified/controlled by an attacker because of a vulnerability, CFI checks to ensure that its unique identifier matches the list of known and allowable target identifiers of the particular instruction [8].

With significant increase in defense solutions against control-flow attacks, exploits which focus on modifying control-flow from memory errors become difficult because these defense solutions have been deployed widely. Alternatively, attacks targeting non-control data do not require changing the application's control flow during an attack. Although such data-oriented attacks can be harmful to systems and can cause significant damage, not many systematic methods to automatically construct them from memory errors have been developed [9]. Attackers usually execute arbitrary malicious code to exploit vulnerabilities in memory. This enables them to use the victim program in order to cause damage to a system and further leak some data from the system. Many of these attacks typically modify a program's control flow by exploiting memory errors. However, such attacks, which focus on modifying control-flow, including code injection and code-reuse attacks, can be prevented by using efficient defense mechanisms such as CFI [10] [11], data execution prevention (DEP), and address space layout randomization (ASLR) [12]. In recent times, such defense solutions have become practical and are being adopted universally in commodity operating systems and compilers, as a result executing control-oriented attacks have become increasingly difficult.

However, attacks which modify control-flow are not the only malicious way to exploit memory errors. Memory errors also enable attacks through corrupting non-control data, a well-known result from Chen et al [13] . In general, non-control data attacks are collectively referred to as data-oriented attacks, which allow attackers to modify the program's data or cause the program to accidentally leak secret/sensitive data. Several recent high-profile vulnerabilities have highlighted the intensity and seriousness of these attacks. In a recent exploit on Internet Explorer (IE) 10,

it has been shown that changing a single byte - specifically the Safemode flag - is sufficient to run arbitrary code in the IE process [9]. The Heartbleed vulnerability is another example where sensitive data from an SSL-enabled server could be leaked without modifying the control-flow of the application [14]. Although data-oriented attacks are very well understood, most of the known attacks are just corruption of non-control data.

Data flow subversion at runtime is a common step of abundant security attacks . Despite previous research on techniques to prevent such attacks, they are still among the most critical security attacks and software is likely to remain vulnerable to them in the future [15]. This can be attributed to the fact that there is a lack of general and platform-independent specification and enforcement of DFI, unclear hypothesis and assumptions and vulnerability-based mitigation techniques. These all result in less precision in the enforcement techniques with possibility of circumvention, and make their evaluations and effectiveness measurements harder [15]. These attacks violate **Data Flow Integrity (DFI)** that imposes restrictions on runtime data flows that are to be allowed by program data flow graph. DFI is firstly defined in [5] informally and its definition was more according to a specific implementation than a general definition. However, previous works do not have a more general implementation independent and platform-independent specification of the policy with explicit assumptions such as an expressive formal study on DFI. In security, to specify policies and evaluate enforcement techniques, it is very important to identify assumptions [15]. Therefore, it is very important and necessary to make proper and clear assumptions and well-defined because an attacker that can invalidate assumptions can also bypass the enforcement. Since in security any set of assumptions is likely to be incomplete, clarifying them makes it simpler to extend or improve the specification and enforcement of a desired policy by completing the list of assumptions, or providing their satisfactions instead of just assuming them in further researches [15].

A brief introduction of the functioning of DFI is given here, in later sections a more detailed explanation of DFI is given. DFI enforces a policy on the data-flow observed at runtime. It ensures that a program must follow a data-flow graph generated via a static analysis at compile time. An instrumentation pass on the program adds checks before each read instruction to ensure that they

do not read a corrupted data. The static analysis uses reaching definition analysis which is a data-flow analysis technique that gives for each read instruction reading a variable, a set of instructions that could have last defined(written to) this variable. For each read instruction the analysis is performed, and each write instruction that defines a variable is assigned an unique identifier. Then, an instrumentation pass adds checks before each write instruction to update a table mapping an address being written to and the last identifier having written a value at that address. In addition, the instrumentation adds checks before each read instruction. It fetches the identifier mapped with the address it is reading the value from, and it ensures that the identifier is in the set of reaching definitions found thanks to the static analysis.

The above mentioned methodology for DFI is the main idea for our hardware based data flow integrity implementation. Using static analysis, we obtain the reaching definition set. The reaching definition set is a set of permissible writes for each memory position which is obtained using the reaching definition analysis. Reaching definition for a given instruction is a previous instruction whose target variable can be assigned to the given one without any assignment in between. After instrumentation of target application, to obtain identifier information, the application is monitored at runtime to ensure it follows the data flow defined by the reaching definition set. Experiments show a performance overhead of about 14.5% for a processor integrated with DFI monitoring system.

2. RELATED WORK

From quite some time, many different methods and ideas for monitoring computer security have been proposed. The basic common principle among all these methods is that they monitor the execution behavior of a program (e.g., control-flow or data-flow) running on the machine to find symptoms of attacks. Among the proposed monitoring schemes, software-based ones are known for their adaptability on the commercial products, but there have been concerns that they may suffer from non negligible runtime overhead [5] [16]. Usually, hardware-based solutions are well known for their high performance. However, most of these hardware solutions have an inherent problem in that they usually introduce drastic changes to the internal processor architecture [17]. More recent ones have tried to minimize such modifications by employing designs with dedicated external hardware security monitors in the system [18] [19]. However, such approaches have some overhead which is caused by communication between the host and the external monitor. Another previous work which focuses on DFI implementation in hardware is the HDFI (Hardware-Assisted Data-flow Isolation). The main objective of HDFI is to prevent malicious attacks from exploiting memory corruption vulnerabilities to tamper/leak sensitive data [20]. This is achieved by making two changes to the design: the ISA extension and the memory tagger. The paper also introduces some optimizations to reduce overhead. HDFI was originally implemented by extending the RISC-V instruction set architecture (ISA) and instantiating it on a Xilinx evaluation board (FPGA) [20].

Initial work in this project involved a solution that relies on external hardware for security monitoring, but unlike the others, this method solves the communication overhead problem by using a dedicated interface called the core debug interface (CDI), which is readily available in most commercial processors for debugging. The CoreSight interface on ARM processors was the interface that was employed as CDI on our initial FPGA setup. The system is built simply by plugging the monitoring hardware into the processor via CDI, precluding the need for altering the processor internals. First, this FPGA prototype setup was used to implement CFI. The experimental results on FPGA prototype showed us promising results, with external hardware monitors efficiently per-

form monitoring tasks with negligible performance overhead. This improved performance can be mainly attributed to use of CDI, which helps reduce communication costs substantially. Later, for the implementation of DFI, a modular discrete event driven computer system simulator, gem5 was chosen. This was done to explore the feasibility and effectiveness of the gem5 platform to perform DFI checking. The focus of the thesis is this implementation of hardware based Data Flow Integrity technique on gem5. The software DFI implementation in [5] is another work which is closely related to the current implementation. More details about the original DFI work [5] and our implementation of hardware-based DFI are discussed in further sections. We give some background about the software DFI paper in Section 3 and discuss our implementation on hardware in Section 4.

3. BACKGROUND

In this section, we give some background about the DFI implementation in [5], as it is the main basis of our work. Then we provide some background about the tools and analysis methods used in this implementation of hardware-based Data Flow Integrity. First a brief introduction to gem5 is given, followed by an overview of Static Value Flow (SVF) analysis tool and then finally the use of instrumentation in this implementation is explained.

3.1 Software data flow integrity

Software data flow integrity was proposed in [5]. The main idea is to have a reference DFG against which the runtime data flow of a program is checked. DFG is obtained using static analysis methods and is used as reference for verifying DFI for an application executed on the processor. According to the implementation proposed by [5], data flow integrity enforcement has three phases. The first phase uses starts with computing a data-flow graph for the vulnerable program, using static analysis. The second phase instruments the program to guarantee that the data-flow at run-time is allowed by the data-flow graph. The third (last) phase runs the instrumented program and raises an exception if data-flow integrity is violated. To enforce data-flow integrity at run-time, this implementation uses instrumentation on the program to compute definition that actually reaches every use at run time. It maintains a table with the identifier of the last instruction to write to each memory position.The table is updated before every write and to prevent the attacker from tampering with the table. Each time a check is performed to find out if the identifier of the instruction that wrote the value being read is an element of the set computed during the static analysis [5]. If it is not, an exception is raised. Taking these implementation details from the software based work, we propose a hardware based implementation in which DFI checking is inbuilt in to the processor and doesn't require any external hardware and interfaces to do so. The software DFI paper [5] proposes some optimizations in software, but all those optimizations haven't been implemented in our design, only the basic idea of data flow integrity is pursued.

3.2 About gem5

Implementation of this DFI checking is done on gem5 simulator. The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. gem5 provides four interpretation-based CPU models: a simple one-CPI CPU, a detailed model of an in-order CPU, and a detailed model of an out-of-order CPU [21]. These CPU models use a common high-level ISA description. gem5 supports multiple ISAs. Any configuration of the above-mentioned CPU models can be used in conjunction with one of the supported ISAs. The current ISAs supported on gem5 are x86, ARM, Alpha, RISC-V, SPARC. We use the simple CPU model and x86 ISA.

To achieve the objectives mentioned above, gem5 source code must be modified. gem5 consists of SimObjects, SimObjects are wrapped C++ objects that are accessible from Python configuration scripts. Python configuration scripts control gem5 and these scripts define the system we want to model. Using these Python configuration files, Simobject parameters are set, these parameters define the processor and memory system configuration and hierarchy. To enable our main objectives of obtaining execution data and performing checks, corresponding Simobjects are modified in the gem5 source code. Most components used in any configuration like CPU, memory buses and memory controllers etc. are MemObjects. The MemObject class extends the ClockedObject and obtains its master and slave ports with the help of accessor functions. The ClockedObject class extends the SimObject with a clock and accessor functions to relate ticks to the cycles of the object [22]. Some important components for understanding gem5 are explained below.

3.2.1 MemObjects

All objects within a memory system inherit from MemObject. The MemObject class adds the pure virtual functions getMasterPort and getSlavePort which returns a port corresponding to the given name. This interface is used to connect memory objects together.

3.2.2 Ports

Ports are used to interface memory objects to each other. They will always come in pairs and we refer to the other port object as the peer. A master port always connects to a slave port, with the master initiating requests, and the slave providing responses. Every memory object has to have at least one port to be useful [22].

3.2.2.1 Atomic/Timing/Functional accesses

There are three types of accesses supported by the ports.

1. Timing - Timing accesses are the most detailed access. They reflect our best effort for realistic timing and include the modeling of queuing delay and resource contention. Once a timing request is successfully sent, at some point in the future the device that sent the request will get a response [22]. We use the timing accesses in our experiments as is it gives us the most practical timing details.

2. Atomic - Atomic accesses are faster than detailed access. They are used for fast forwarding and warming up caches and return an approximate time to complete the request without any resource contention or queuing delay. When an atomic access is sent the response is provided when the function returns [22].

3. Functional - Like atomic accesses functional accesses happen instantaneously, but unlike atomic accesses they can co-exist in the memory system with atomic or timing accesses. Functional accesses are used for things such as loading binaries, examining/changing variables in the simulated system, and allowing a remote debugger to be attached to the simulator [22].

3.2.3 Packets

A Packet is used to encapsulate a transfer between two objects in the memory system . This is in contrast to a Request where a single Request travels all the way from the requester to the ultimate destination and back, possibly being conveyed by several different packets along the way

[22]. Read access to many packet fields is provided via accessor methods which verify that the data in the field being read is valid. A packet contains the following all of which are accessed by accessors to be certain the data is valid:

- The address. This is the address that will be used to route the packet to its target and to process the packet at the target. It is typically derived from the request object's physical address.
- The size. Again, this size may not be the same as that of the original request, as in the cache miss scenario.
- A pointer to the data being manipulated
  - get() and set() methods are used to manipulate the data in the packet.
- A list of Packet Command Attributes associated with the packet
- A pointer to the request

3.3 Static Value Flow (SVF)

- SVF: SVF is a tool that enables scalable and precise inter procedural Static Value-Flow analysis for C programs by leveraging advances in sparse analysis [23].
- LLVM : LLVM is a library for programmatically creating machine-native code. A developer uses the API to generate instructions in a format called an intermediate representation, or IR. LLVM can then compile the IR into a standalone binary, or perform a JIT (just-in-time) compilation on the code to run in the context of another program, such as an interpreter for the language [24].
- Clang : It is a front end to the LLVM compiler and is designed to compile C, C++, Objective-C, and Objective-C++ to machine code. Apple is the primary developer of clang [25].

Static value-flow analysis resolves both the data and control dependences of a program. It was initially adopted in software debugging [26], [27] and optimising compilers [28], [29] by providing explicit definition-use relations of program variables. This fundamental technique has subsequently been used widely for program analysis and verification in many open-source and commercial tools [23].

SVF is a static tool that enables scalable and precise interprocedural dependence analysis for C and C++ programs. SVF allows value-flow construction and pointer analysis to be performed iteratively, thereby providing increasingly improved precision for both [23].

To find the reaching definition set for an application, SVF is used. Initial work used Program Counter as identifier for an instruction, but for larger applications a better way to find reaching definition set was required. SVF is a static tool that enables scalable and precise interprocedural dependence analysis for C and C++ programs [23]. SVF constructs reaching definition set using Node ID as identifier for instructions. Implementation was modified accordingly to perform checking using Node ID instead of Program Counter. In order to relate an instruction (in our case load/store) with Node ID, we use code instrumentation. We use the information obtained through Code Instrumentation and generate instrumented binaries/executable. Reaching definition set and instrumented binary are inputs to perform checking. Reaching Definition Set acts as reference/specification for the DFI checking. Executable is application to be run on CPU model on gem5 or the workload for the gem5 simulation.

3.4 Instrumentation

Instrumentation is a process through which certain instructions are inserted to an existing executable/program to obtain extra internal information from the program, in order to have a better understanding about the execution of the program. Instrumentation is broad term which can incorporate code tracing, profiling, debugging/exception handling, performance counters and data logging. In this implementation, retrieving the unique instruction identifier is very important as the identifier is used for checking the dynamic data flow of an application/program at run time. As previously mentioned, initial implementation used PC as the identifier but this was good only for

small programs where DFG could be constructed manually. But for larger applications, SVF analysis is used for constructing the reaching definition set. The tool uses Node IDs as the identifier for instructions and constructs the RDS. Corresponding changes were made to the DFI checking logic in gem5. To accommodate for these changes, the Node IDs should be able to be retrieved from the binary(application) at run time. This is solved by instrumenting the program binary and inserting instructions in to the binary such that the identifier can be obtained by the DFI checking logic in gem5.

4. IDEA & IMPLEMENTATION

4.1 Objective

The main objective of this thesis is to present a simple hardware based technique that prevents control and non-control data attacks by enforcing data flow integrity. It computes a data flow graph using static analysis, and it instruments the program to ensure that the flow of data at runtime is allowed by the data flow graph [5]. As with any real time verification system the main steps involved are 1) to collect and store trace data without loss (information useful to perform checking) and 2) use the data and perform relevant checking (like control-flow or data-flow). In this section, first a brief idea about the implementation is given, then the method of static analysis we followed is explained. Further, the concept of DFI is explained with a simple example and then the implementation of the DFI checking on gem5 is presented.

4.2 Idea

The main idea is to have a reference DFG against which the runtime data flow of a program is checked. DFG is obtained using static analysis methods and is used as reference for verifying DFI for an application executed on the processor.

- Information/data extraction from the processor: One of the most important aspect for performing hardware based DFI or CFI is to obtain run time execution data from the processor. Execution data like Program Counter, data in each instruction and address of the data in an instruction are very crucial for the verification to be done. After acquiring the data required for checking, the next step is to store/move data without any loss. In case of an FPGA, trace data is obtained through a debug module like ARM CoreSight or Intel PT, which provide a wide range of operations to get execution time trace data. They also have specialised interfaces like TPIU (Trace Port Interface Unit) on CoreSight, which is used to connect and transfer trace data to external modules. For gem5, run time data is obtained by modifying source code to place data on the bus and makes it available to the module which needs this

data to perform DFI checking.

Once the data required for checking is available, depending on the type of instruction (read or write) different course of actions are taken to perform DFI checking. This data could be fetched from memory or is directly fetched from the bus when it is being transmitted to another unit, it depends on the data we want. As part of instrumentation, identifiers for every read and write instructions are stored and can be accessed at run time to check for the intended data flow using the reference DFG/Reaching definition set (RDS) obtained using static analysis.

- For every store (write), the identifier for that instruction is stored in a dynamic table (called reaching definition table). This table is updated with the identifier for the most recent write for every variable. This dynamic table is stored in memory.
- For a load (read) instruction, whenever a value is read, the identifier of that read instruction is looked up in the reference table. Then the latest write identifier to that variable (which is stored in Reaching Definition Table) is searched for in the set of valid writes to that variable. If the identifier is present in the RDS, then the data flow integrity is not violated, if not then there is a DFI violation.

4.3 Static Analysis

Static analysis is used to obtain the reaching definition set, which gives us the set of allowable writes when a particular variable is read. This is done offline and the product of the static analysis is used as reference data for the runtime execution of DFI checking performed on gem5.

The main steps involved in static analysis are:

1. SVF analysis
    - Generate bitcode file
        - Convert source to bitcode: The source needs to be compiled with Clang to generate the bitcode(.bc) files

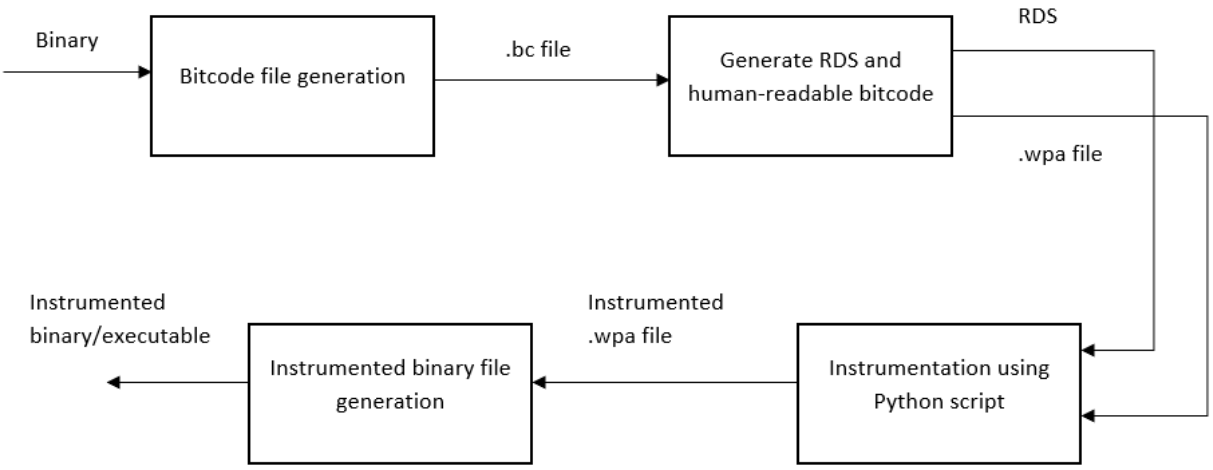

Figure 4.1: Flow Chart for obtaining instrumented binary

  - Link bitcode files: Links all (.bc) file into a large single (.bc) file through LLVM Gold plugin

- Generate Reaching Definition Set
  - By using options present in the SVF tool, we obtain identifiers (called Node IDs in this case) of instructions and reaching definition set for the program using these Node IDs as identifiers. In this stage, we also generate the .wpa file.

2. Instrumentation

- We next obtain the wpa.ll file on which instrumentation is performed. The purpose of instrumenting the program is to get the Node ID of every load and store instruction, these Node IDs are used as identifiers later for DFI checks at execution time
- The number of instructions in a small program are usually low and instrumentation for these programs can be performed manually, but for a benchmark which has millions of instructions it is not feasible or sensible to perform manual instrumentation. Therefore, a python script is used which uses the generated reaching definition set and the wpa.ll

file as input to generate an instrumented .ll file

3. Executable/binary generation

    - Using llvm options, a new bitcode file is generated for the instrumented .ll file which is further used to generate the instrumented executable file.

This process is followed for every program/application/benchmark and the flowchart in Figure 4.1 shows a concise flowchart to explain this process.

4.4 Detailed DFI Explanation

The idea of Data flow integrity as proposed in [5] is explained with an example. Figure 4.2 shows a simple piece of code which will be used to explain the idea of data flow integrity. The sample code has a few reads and writes to certain variables x, y and z. Values are written for variable x on lines 1 and 5 and for variable y on line 2. The value of x is read on line 8 and z is written on line 8. The illegal operation on line 6 represents a malicious write on x inserted at run time. This code has two execution paths because of the if loop. Figure 4.3 shows the data flow graph for the sample code. D1 is the identifier for the write on x in line 1, D2 is the identifier for the write on y in line 2, D3 is the identifier for write on x in line 5 and D4 is the identifier for write on z in line 8. These identifiers are important for the construction of a DFG and the reaching definition set. The reaching definition set acts as a reference for the dynamic DFI checking. Another important component for DFI checking is the Reaching Definition Table (RDT) which holds the identifier for the most recent write to a particular memory address (variable). Figure 4.4 shows enforced DFG/Reaching Definition Set.

```
1 x=100;
2 y=10;
3 if(...)
4 {
5     x=200;
6     //illegal.memory_op
7 }
8 z = x*10;
```

Figure 4.2: Sample code

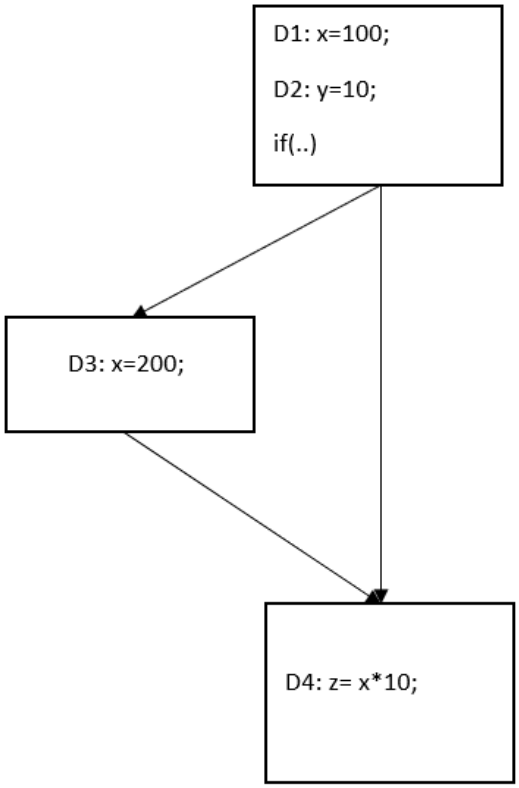

Figure 4.3: Data Flow Graph for the sample code

```
x -> {D1,D3}
y -> {D2}
z -> {D4}
```

Figure 4.4: Enforced (allowed) Data Flow for each variable

In Figure 4.5, the execution path (Execution Path 1) is shown for when the ‘if’condition is not met. In this path, we have a read on variable x in line 8. The most recent write to x is on line 1 i.e D1. D1 is present in the reaching definition set in Figure 4.4, hence this is a valid scenario and is allowed by the DFG.

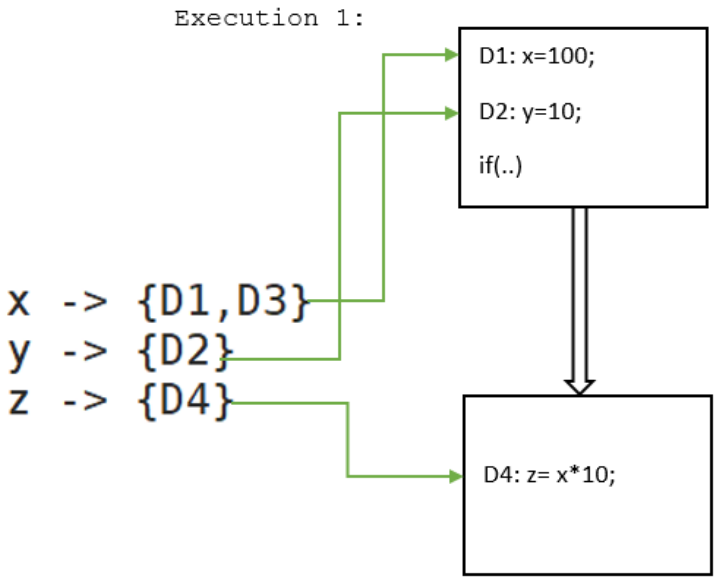

Figure 4.5: Execution Path 1 for sample code

Next, we look at the other execution path (Execution Path 2) shown in Figure 4.6, when the ‘if’condition is met. Here, the writes on X occur at lines 1(D1), 5(D3), 6. But the write at line 6 is not allowed by the reaching definition set, represented by the identifier D_illegal. This is the most

recent write on x before being read on line 8 but as it is not present in the reaching definition set, it is an illegal operation. As a result, we say that Data Flow Integrity is violated in this case.

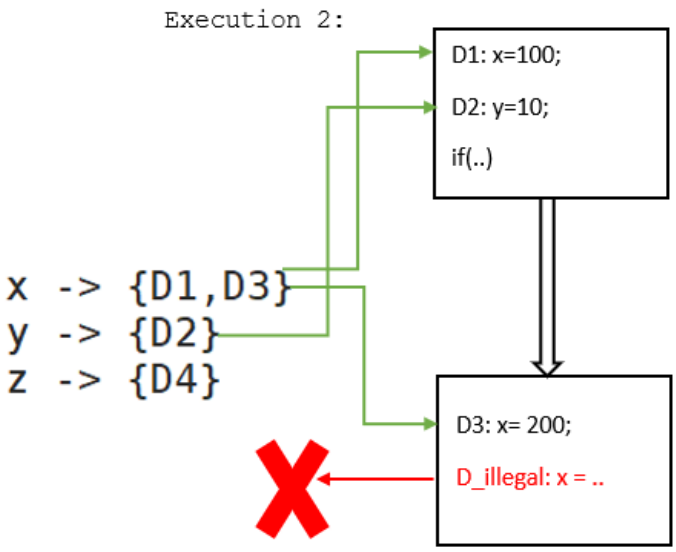

Figure 4.6: Execution Path 2 for sample code

4.5 Hardware-based DFI Checking

In this section, the implementation of the DFI checking mechanism introduced in section 4.4 is explained with respect to the gem5 simulator. Even though the main idea is the same, the way data required to perform DFI is obtained and how it is stored and used changes. As shown in Figure 4.1, the instrumented binary and reaching definition set are obtained through static analysis and these files are used as inputs for simulations in the gem5 environment (Figure 4.7).

The changes that have been made in the gem5 simulator source code can be mapped roughly to some components on a real processor. As major changes in the source code have been done in the read and write handling methods, the changes on a processor would be mapped to a load-store unit, which is a specialized execution unit which is responsible for the execution of all load and store instructions. In addition, for any arithmetic operations required for checking can be performed in

the integer execution unit or the ALU. Hence, most of the changes in this implementation are in the load/store unit or the ALU, when mapped to real hardware on CPU. Further, the current system configuration is very basic and doesn't have out of order execution, a branch predictor, a prefetcher or a cache memory hierarchy. In a system with all the previously mentioned components, the overhead could be further reduced.

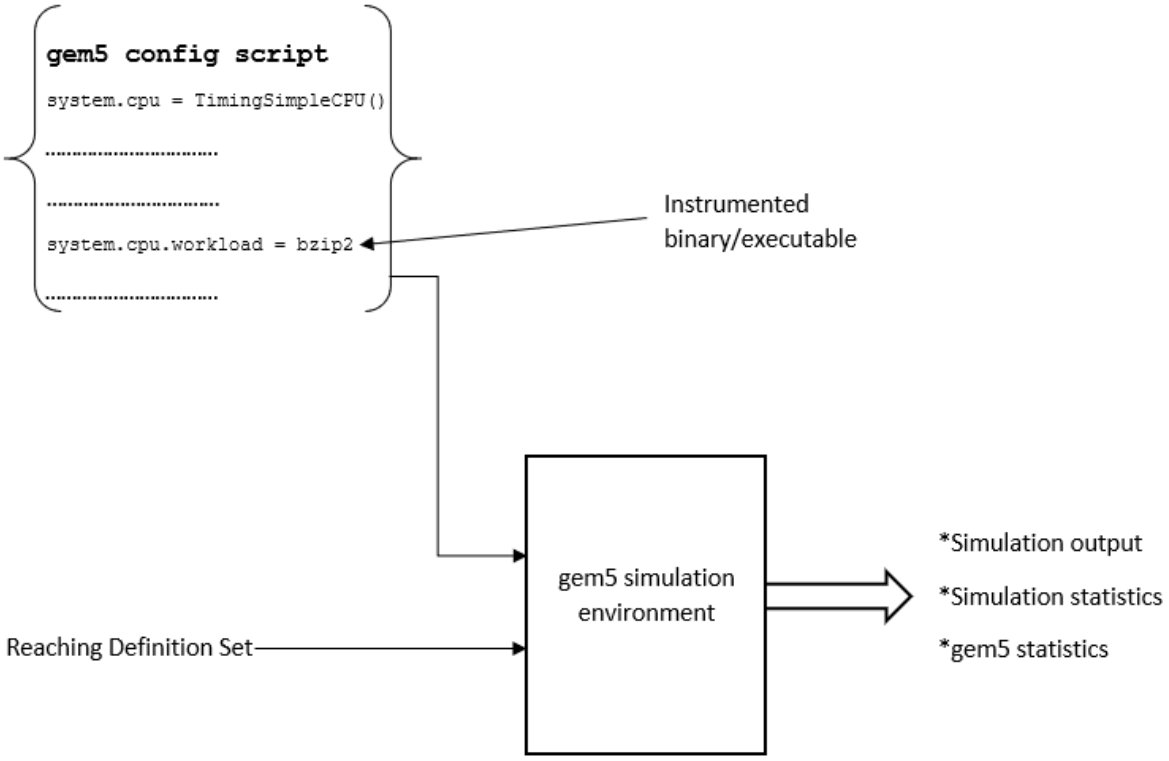

Figure 4.7: Simulation environment for gem5

gem5 has a flexible hierarchy and structure for CPU, Memory system, Bus and other peripherals. For designing different CPU models, there are a wide range of options available on gem5 but for adding extra checking logic like the one we need for DFI or CFI, the source code has to be modified. Going in to the details of the implementation, the DFI verification logic has to be added at the most appropriate location in the gem5 source code. The most challenging part about modifying

gem5 source code is to explore the source code and find the right place to add the checking logic, without disturbing the normal functioning of the gem5 environment. Initial trials involved trying to obtain run time information like Data, address, Program counter and instruction type from existing modules/objects on gem5, but this information couldn't be transferred over different memory objects for reuse. Then, the source code was modified in accordance with the requirements we need for obtaining and using the run time information from the system. The verifying code is added in the CPU module and the read and write functions which process read and write packets in a gem5 simulation are the main targets for code addition/modification.

In this implementation, we use reaching definitions analysis to construct a reaching definition set, which acts as a reference for the run-time checking. This file is loaded at run time and used for every future read and write reference. For each value read by an instruction, a set of instructions that may write the value are computed. The analysis relies on the same assumptions that existing compilers rely on to implement standard optimizations. These are precisely the assumptions that attacks violate and data-flow integrity enforcement detects when they are violated [5]. Next, a reaching definition table is used to store the identifier (Node ID in this case) for the most recent write to every address (variable). Initially, the program counter was used as the identifier for instructions, but this was feasible only for smaller programs. Subsequently, for running larger workloads and constructing reaching definition set for these programs we used SVF analysis and instrumentation. Using the idea mentioned in Section 4.2, the Reaching Definition Table and the Reaching Definition Set are used to implement the DFI checking. The main logic of checking remains the same as the example explained in Section 4.4. Whenever there is a store instruction, the RDT is updated for that variable with the latest identifier. When there is a load instruction, the the identifier present in the RDT for that variable is picked and it is looked up in the RDS. If the identifier is not present, a Data Flow violation has occurred.

5. EXPERIMENTAL SETUP & RESULTS

5.1 Experimental Setup

As mentioned earlier, we use the gem5 simulator to perform DFI checking and execute simulations to obtain different parameters. The system configuration for the gem5 CPU model used for simulations is shown in Figure 5.1. We use a TimingSimple CPU model as defined in gem5. The TimingSimpleCPU is the version of SimpleCPU that uses timing memory accesses. It stalls on cache accesses and waits for the memory system to respond prior to proceeding [30]. The TimingSimpleCPU is derived from the BaseSimpleCPU, and implements the same set of functions. It defines the port that is used to hook up to memory, and connects the CPU to the cache. It also defines the functions which are required for handling the response from memory to the accesses sent out. This CPU model can be used with or without a cache hierarchy. In the absence of caches, the CPU ports for icache (Instruction Cache) and dcache (Data Cache) are directly connected to a bus. In our case, the dcache port from CPU is connected to a memchecker. The memchecker provides debug options for reads and writes in the processor and proves helpful for debugging any errors on the data port of the CPU and it also provides additional information for reads and writes. A system wide memory bus is used for communication between the CPU and memory. Next, a memory controller is created and connected to the memory bus. For this system, a simple DDR3 controller is used and it is responsible for the entire memory range of the system.

The parameters set for the experimental setup are shown below in Table 5.1. Simulations are

gem5 CPU Model	Timing Simple
Width	64 bits
Memory Hierarchy	No caches used
ISA	x86
Frequency	1GHz/2GHz
gem5 Memchecker	Used on data port

Table 5.1: gem5 system configuration

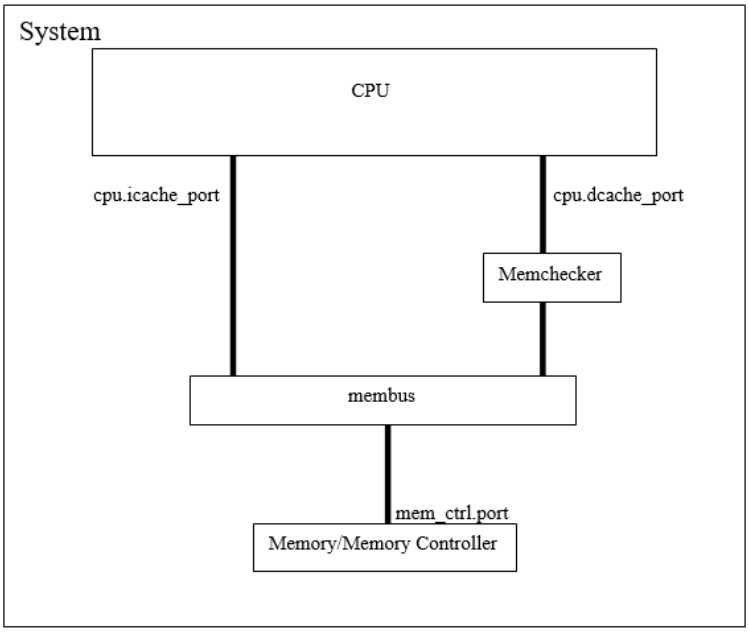

Figure 5.1: gem5 configuration used

run on SPEC CPU2006 benchmark suite. The SPEC CPU 2006 benchmark is SPEC's industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler [31]. This benchmark suite consists of the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 different benchmark tests and the SPECfp 2006 benchmark contains 19 different benchmark tests. We select 6 integer benchmarks shown in Table 5.2 from SPEC CPU 2006 suite and run simulations on these benchmarks [31]. The limit on number of benchmarks used for simulations is that the SVF tool cannot compute the reaching definition set and generate the bitcode file which is used for obtaining the instrumented executable for a benchmark.

Benchmark	Application	Description
401.bzip2	Compression	Performs compression and decompression on inputs at different compression levels
445.gobmk	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game
456.hmmer	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
462.libquantum	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm
464.h264ref	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets
429.mcf	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm to schedule public transport

Table 5.2: Benchmarks with descriptions

5.2 Results

Using the setup and benchmarks mentioned in Section 5.1, simulations were run on gem5 for different number of instructions to get a good idea about the time overhead, instruction type behaviour over a different range. These experiments were done for both, an unmodified gem5 CPU and the modified gem5 CPU which has the DFI checking built-in. Figures 5.2 - 5.6 show the results for all the simulations run on different benchmarks. As expected, there is some time overhead in case of the modified CPU because it performs additional DFI checking on every read and write. For every benchmark and for the different number of instructions simulated, we see that there is some increase in simulated time. The difference between the unmodified and modified CPU is measured to calculate the time overhead. The time overhead on an average comes to up to 14.5%.

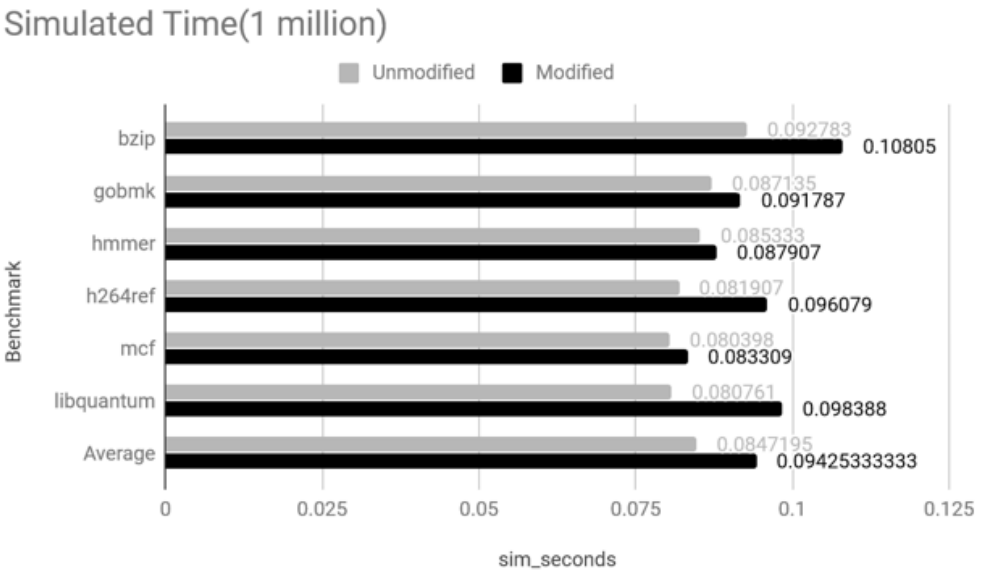

Figure 5.2: Simulated time comparison for 1 million instructions

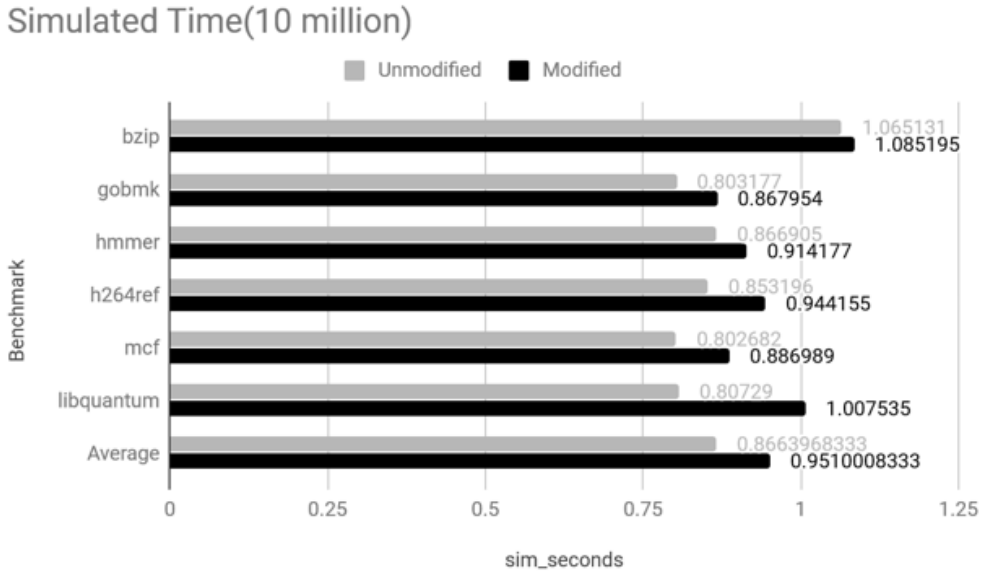

Figure 5.3: Simulated time comparison for 10 million instructions

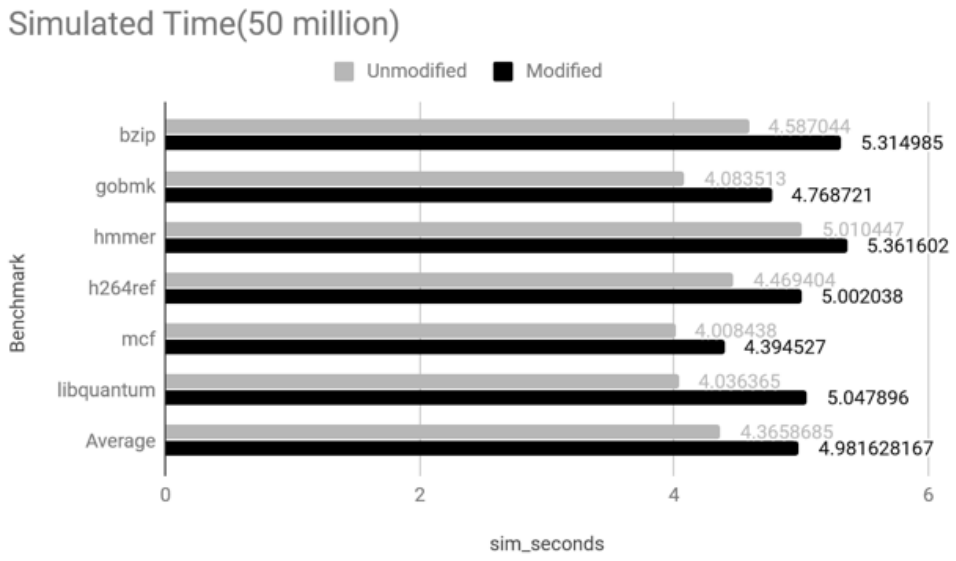

Figure 5.4: Simulated time comparison for 50 million instructions

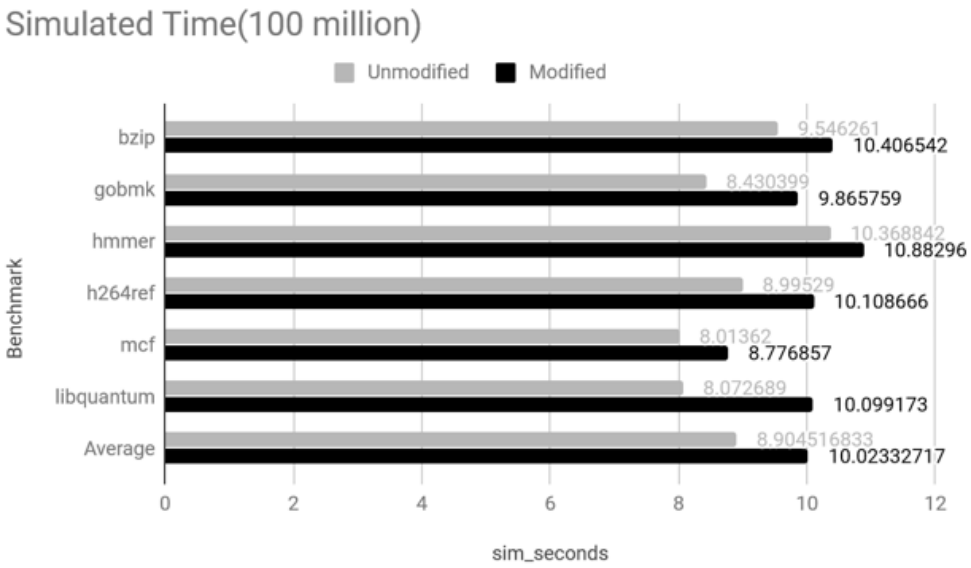

Figure 5.5: Simulated time comparison for 100 million instructions

A summary of all the simulations and the time overhead for the modified gem5 CPU is presented in Table 5.3. The columns are for the the number of instructions simulated (1 million, 10

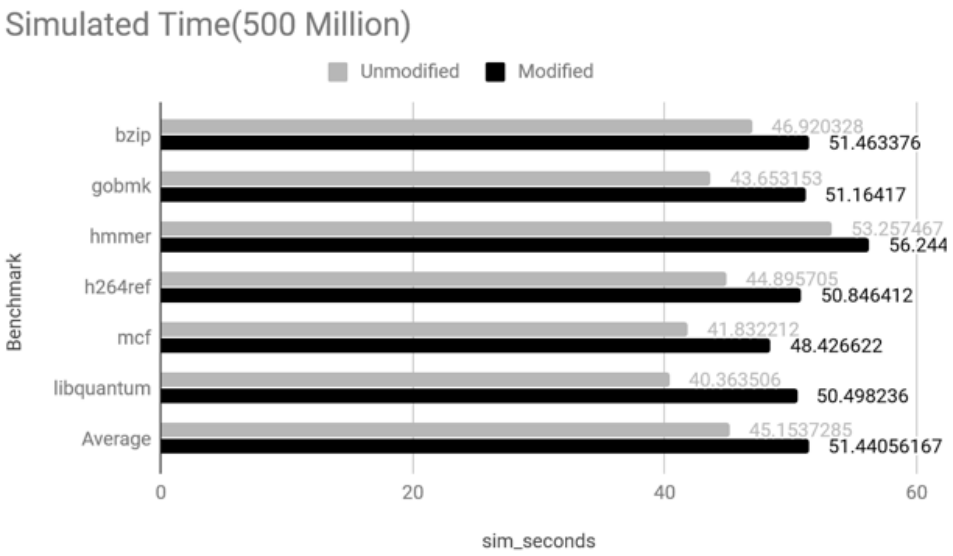

Figure 5.6: Simulated time comparison for 500 million instructions

million etc.) and the corresponding values for each benchmark shows the percentage increase in time to simulate the respective number of instructions on the modified gem5 CPU.

	1mn	10mn	50mn	100mn	500mn
401.bzip2	16.45	1.88	15.87	9.01	9.68
445.gobmk	5.34	8.06	16.78	17.03	17.21
456.hmmer	3.02	5.45	7.01	4.96	5.61
462.libquantum	21.83	24.80	25.06	25.10	25.11
464.h264ref	17.30	10.66	11.92	12.38	13.25
429.mcf	3.62	10.50	9.63	9.52	15.76
Average	11.26	10.23	14.38	13.00	14.44

Table 5.3: Overall average time overhead(% increase for modified CPU)

As mentioned in Section 3.3, our implementation uses instrumentation to obtain information about the identifiers for every load and store. For a better understanding of the overhead caused by the modifications made to gem5 source code, further analysis of timing information was performed. As mentioned earlier, the handleWritePacket method in the CPU source code contains the most

crucial changes for the DFI checking. Hence, we measured the average simulated time it takes to perform a write taking into account the extra checking code. The simulations were repeated for all benchmarks for CPU models with and without the modifications and the average simulation ticks were measured for executing the write method. Average percentage increase in simulation ticks for the write method is shown in Table 5.4.

Benchmark	% increase
401.bzip2	46.94
445.gobmk	133.09
456.hmmer	25.51
462.libquantum	0.46
464.h264ref	19.60
429.mcf	19.27
Average	40.81

Table 5.4: Average time overhead for Write (% increase for modified CPU)

In addition to the measuring the time overhead, we also compare the number of writes for each benchmark in both the cases (modified and unmodified CPU). This is made to check the effect of instrumentation in the case of modified CPU, for which instrumented benchmark binaries are used. As explained in previous sections, the instrumentation in our case adds additional writes (stores) to the binary to store information of the instruction identifier for every read and write. As a result, we expect the number of writes to increase in the simulations for the instrumented binaries that are run on the modified CPU. In the figures 5.7 to 5.11, the number of writes for both the cases are shown. As we can observe in the figures below, for simulations of any number of instructions and for every benchmark, we see an increase in the number of writes as expected. This, as mentioned before, is due to the fact that we instrument the program binary with store instructions.

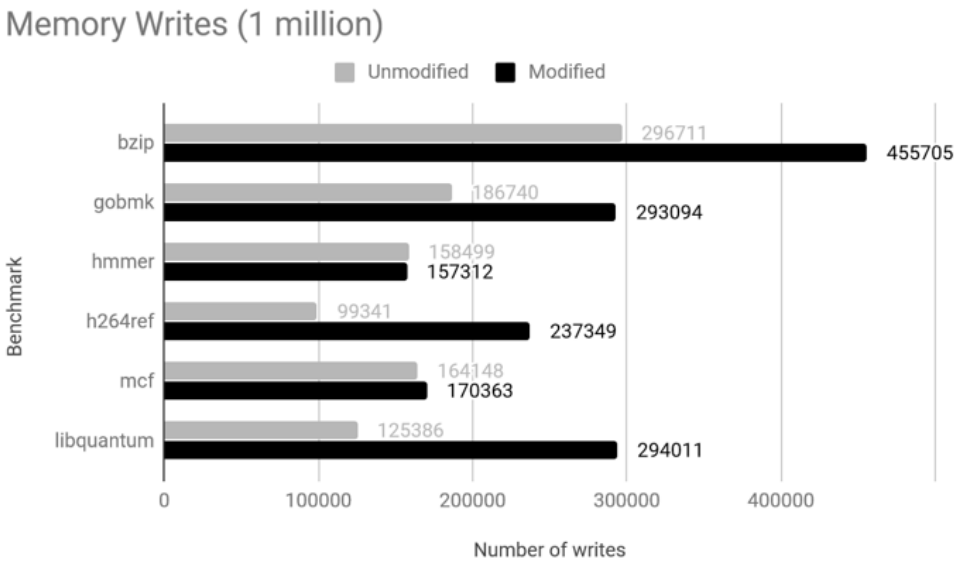

Figure 5.7: Comparison of number of writes for 1 million instructions

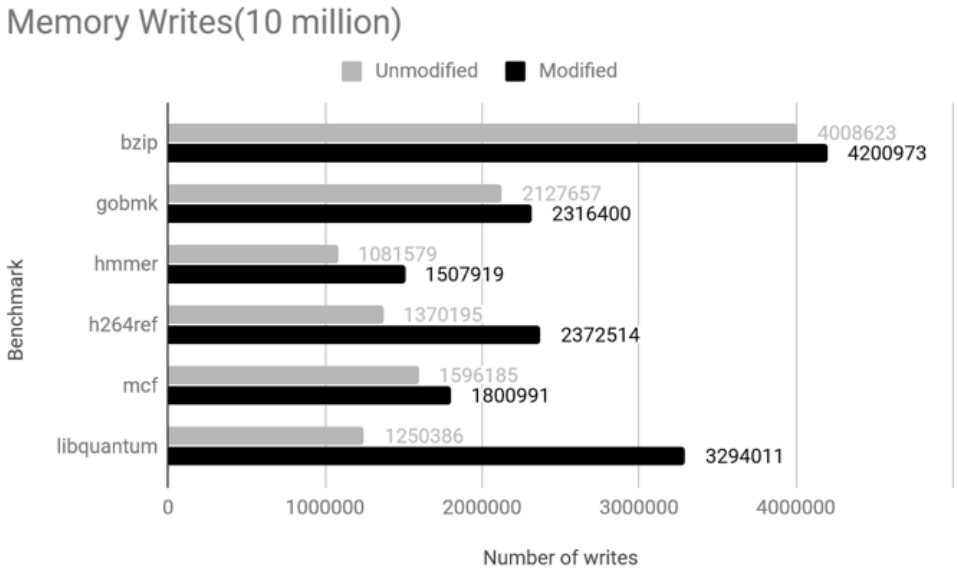

Figure 5.8: Comparison of number of writes for 10 million instructions

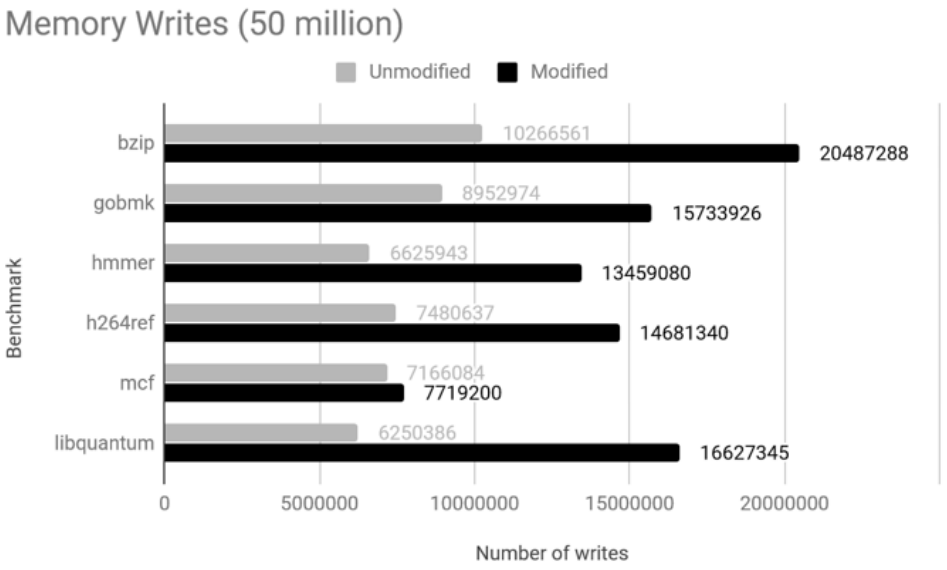

Figure 5.9: Comparison of number of writes for 50 million instructions

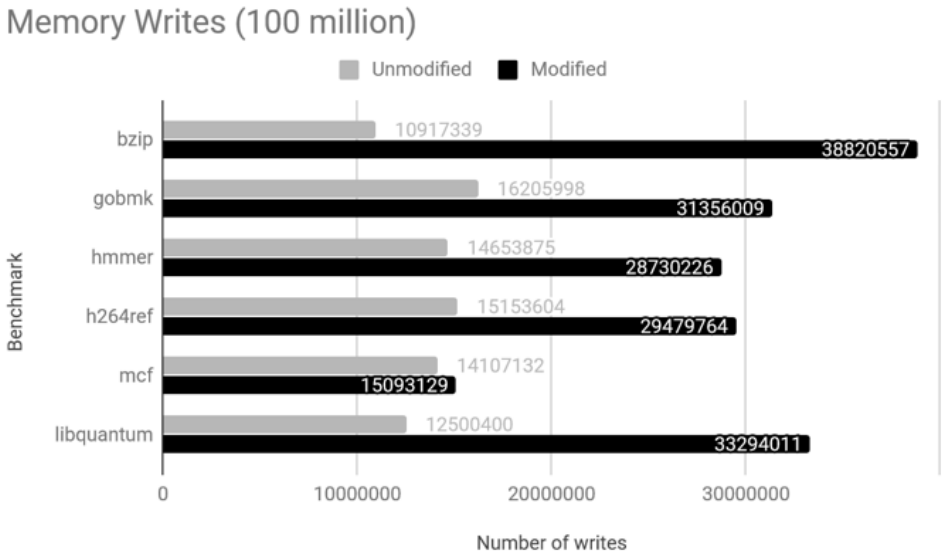

Figure 5.10: Comparison of number of writes for 100 million instructions

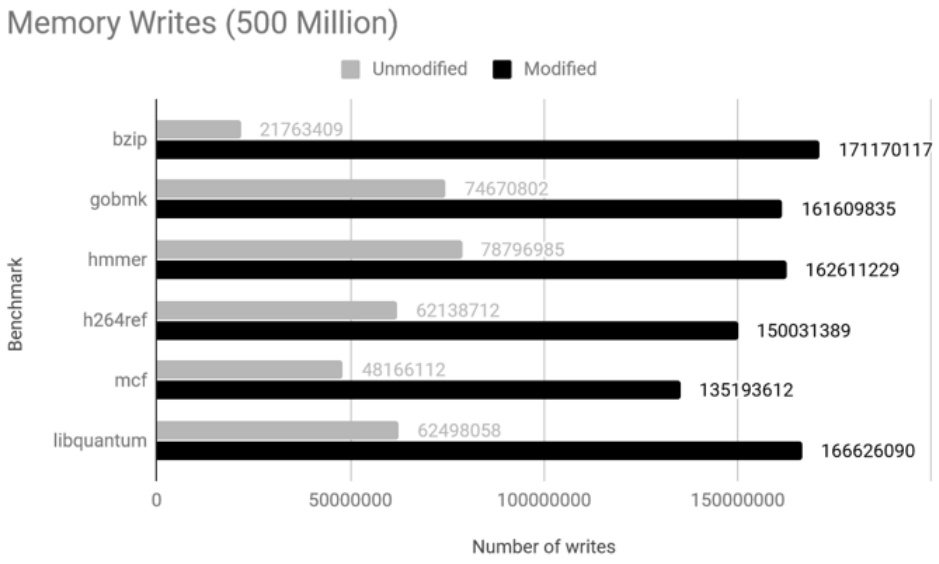

Figure 5.11: Comparison of number of writes for 500 million instructions

In table 5.5, a summary is provided to show the percentage increase in number of writes for each benchmark and for simulations run for different number of instructions (mn in the table represents millions). The overall trend is that as number of read, writes increase the the number of writes increase in the case of the instrumented binary as a result of the extra stores used to store the information of the instruction identifier for DFI checking.

	1mn	10mn	50mn	100mn	500mn
401.bzip2	53.58	4.80	99.55	255.58	686.50
445.gobmk	56.95	8.87	75.74	93.48	116.43
456.hmmer	-0.75	39.42	103.13	96.06	106.37
462.libquantum	138.92	73.15	96.26	94.54	141.44
464.h264ref	3.78	12.83	7.72	6.99	180.68
429.mcf	134.48	163.44	166.02	166.34	166.61
Average	64.50	50.42	91.40	118.83	233.00

Table 5.5: Summary of number of writes(% change for modified CPU)

Even though this implementation was done on gem5 simulator, we try to provide an idea about the extra hardware that might be required for making these changes on FPGA hardware. For this we use previous knowledge from our implementations of similar checking methodologies on FPGAs. In a collaborative work where we implemented CFI on an Altera FPGA, the required number of ALMs were in the range of 18,000-32,000 for different benchmarks. ALM stands for adaptive logic module in Altera FPGA, which is the basic element of FPGA and similar to LUT (Lookup Tables). As the checking logic for DFI is of similar complexity to CFI, we estimate the hardware requirement to be in the order of tens of thousands ALMs. Fur further understanding, we also estimate the approximate number of additional gates required for our implementation. We take the maximum count of 32,000 LUT/ALMs for this estimation. As per the calculations in [32], every LUT is equivalent to 6 2-input ASIC NAND gates. With this assumption, we calculate the number of gate required for our design as 32000*6, which is equal to 192,000 (nearly 0.2 million). In order to compare how this number stands against number of gates in a typical modern processor, we compare it with a Intel Core i7 processor (nearly 450-500 million gates). The number of gates in our design is about 0.2 million and as a percentage increase when compared with a typical processor, it comes to about 0.044%. Please note that we have taken a conservative number of around 450-500 million gates in a modern processor, processors these days have a much higher number of gates and as result, our design will account for a lower percentage than the figure(0.044%) mentioned above. In addition, we estimate the memory requirement of storing the Reaching definition set and reaching definition table to be in the order of hundreds of megabytes.

Owing to differences in hardware,operating system and compiler, it is not very straightforward to perform a detailed quantitative comparison with existing techniques. But we can use published results obtained using the same benchmark suite (SPEC) to put our overhead in perspective. As mentioned earlier, the overall average overhead for our implementation is about 10-15% for the selected benchmarks, and the average overhead due to the DFI checking logic is 40.8%. We compare the overhead numbers of a few previous works to our implementation, keeping in mind that these may be not be very accurate because of the factors mentioned above. The software

DFI implementation in [5] introduces two variants of DFI, interproc DFI and intraproc DFI. As per the published results, the average overhead for interproc DFI is 104% and for intraproc DFI, it is 43%. Further, other published results from older works are shown in [5]. Compared to the software DFI overhead, Program Shepherding [33] and CFI[34] have lower overhead but these techniques fail to detect data-oriented attacks. The overhead of either variant of software DFI is significantly lower than the overhead incurred by a state-of-the-art C bounds checker:CRED [35], which incurs an overhead of nearly 300% in bzip2 and 100% in gzip [35].The overhead of software implementations of taint checking [16] [36] is also significantly higher, for example, TaintCheck [16] ran bzip2 37.2 times slower than without instrumentation [5].

Another previous work, HDFI [20] by Song et al., takes a hardware based approach to data flow isolation. The goal of HDFI is to prevent attackers from exploiting memory corruption vulnerabilities to tamper/leak sensitive data. To achieve this goal, they leverage data-flow integrity [5]. The average overhead for their implementation is 21.7%, which after optimizations can be reduced to about 1-2%. These numbers presented above are just to give a perspective of overhead in different previous related works, the implementation details and methodologies in each of them is different but the basic idea of protecting software from security attacks remains the same.

6. CONCLUSION

Overall, from the results obtained from the experiments run on the modified CPU supporting DFI checking, we can conclude that the changes introduced have a certain time overhead which is of the order of 10-15%. Focusing closely on only the overhead caused by writes (as write method has the DFI verification logic), we see that the overhead is roughly about 40% as compared to the gem5 CPU system with no changes. As expected, modified CPU system has more writes because of the code instrumentation and DFI checks on every load and store instruction in the CPU but will make sure that data flow integrity is not violated in the application (benchmark in this case). Also it is estimated that this can be achieved with a decent amount of extra hardware cost. For this, we provide an estimated based on both approximate number of LUTs and gates that may be required when our design is implemented in real hardware.

REFERENCES

[1] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.

[2] "President's information technology advisory committee, cybersecurity: A crisis of prioritization, executive office of the president, national coordination office for information technology research and development." `https://www.nitrd.gov/Pitac/reports/20050301_cybersecurity/cybersecurity.pdf`, 2005.

[3] J. Ransome and A. Misra, "The importance of software security." `http://www.ittoday.info/ITPerformanceImprovement/Articles/2013-12RansomeMisra.html`, 2013.

[4] K. Rahimunnisa, A. S. A., and K. T.S., "Hardware assisted address monitoring system," in *IEEE International Conference on Computer Engineering and Applications*, vol. 1, pp. 537–541, 2010.

[5] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *USENIX Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006.

[6] Wikipedia contributors, "Runtime verification — Wikipedia, the free encyclopedia," 2019.

[7] Wikipedia contributors, "Control-flow integrity — Wikipedia, the free encyclopedia," 2019.

[8] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy*, pp. 575–589, 2014.

[9] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Conference on Security Symposium*, pp. 177–192, 2015.

[10] M. Abadi, M. Budiu, and U. Erlingsson, "Control-flow integrity," in *ACM Conference on Computer and Communication Security*, pp. 340–353, 2005.

[11] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE Symposium on Security and Privacy*, pp. 559–573, 2013.

[12] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *USENIX Conference on Security Symposium*, 2003.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Conference on Security Symposium*, pp. 12–12, 2005.

[14] Wikipedia contributors, "Heartbleed — Wikipedia, the free encyclopedia," 2019.

[15] T. Ramezanifarkhani and M. Razzazi, "Principles of data flow integrity: Specification and enforcement," *Institute of Information Science Journal of Information Science and Engineering*, vol. 31, pp. 529–546, 03 2015.

[16] J. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *The Internet Society Annual Network and Distributed System Security Symposium*, 2005.

[17] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *ACM/IEEE Annual International Symposium on Computer Architecture*, pp. 482–493, 2007.

[18] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient security monitoring with the core debug interface in an embedded processor," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 8:1–8:29, 2016.

[19] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *IEEE International Conference on Dependable Systems Networks*, 2009.

[20] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *IEEE Symposium on Security and Privacy*, pp. 1–17, 2016.

[21] J. Lowe-Power, "Simobjects in the memory system." http://learning.gem5.org/book/part2/memoryobject.html. gem5 Tutorial 0.1 documentation.

[22] "General memory system." http://www.gem5.org/General_Memory_System. gem5 source code documentation.

[23] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *ACM International Conference on Compiler Construction*, pp. 265–266, 2016.

[24] "The llvm compiler infrastructure." https://llvm.org/. Maintained by the llvm-admin team.

[25] M. Wilson, "An introduction to clang." https://www.ics.com/blog/introduction-clang, 2013.

[26] M. Weiser, "Program slicing," in *IEEE International Conference on Software Engineering*, pp. 439–449, 1981.

[27] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[28] B. Steffen, J. Knoop, and O. Rüthing, "The value flow graph: A program representation for optimal program transformations," in *European Symposium on Programming*, 1990.

[29] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[30] "Simplecpu." http://www.m5sim.org/SimpleCPU. gem5 Tutorial 0.1 documentation.

[31] "SPEC CPU 2006." https://www.spec.org/cpu2006/. Copyright 1995 - 2019 Standard Performance Evaluation Corporation.

[32] M. Posner, "How many ASIC gates does it take to fill an FPGA." https://blogs.synopsys.com/breakingthethreelaws/2015/02/how-many-asic-gates-does-it-take-to-fill-an-fpga/, 2015.

[33] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *USENIX Conference on Security Symposium*, pp. 191–206, 2002.

[34] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, 2009.

[35] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *The Internet Society Annual Network and Distributed System Security Symposium*, pp. 159–169, 2004.

[36] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *ACM Symposium on Operating Systems Principles*, pp. 133–147, 2005.